Survey on Subtree Matching

Péter Cserkúti, Tihamér Levendovszky, Hassan Charaf
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Műegyetem rkp. 3-9, H-1111 Budapest
Hungary
csiri@aut.bme.hu, tihamer@aut.bme.hu, hassan@aut.bme.hu

***Abstract* – The subtree matching is the problem of finding all the matching occurrences of a pattern tree in a subject tree as a subtree. This paper introduces many problems that are related to subtree matching and provides a classification of them. The basic problem has got several variations depending on the type of trees and on what we call a match. Trees can be rooted or unrooted, ordered and unordered and labeled or unlabeled. A subtree matching may mean subtree isomorphism, subtree homeomorphism, subtree inclusion or pattern matching. This paper introduces the most important algorithms for these problems while paying special attention to their complexities.**

I Introduction

The different kinds of subtree matching problems have got great importance in different fields of science. This paper provides an overview and a classification of these problems and the algorithms that solve them. There is a great variety of applications that uses these techniques. They can be suitable for code generation, searching in hierarchical data structures for example when executing XML queries [14] or it can also be used in bioinformatics and chemistry. Subtree matching is a special case of subgraph matching. Subgraph isomorphism is NP-hard [9][10], since it includes, for example, the clique problem. While subexponential algorithms are unlikely in the case of graphs, subtree isomorphism can be solved in polynomial time. However, isomorphism is not the only approach of subtree matching as it will be revealed later. While subtree matching is a subset of subgraph matching, it is a generalization of string pattern matching problems as well. The difference is the fact that while strings are linear structures trees are non-linear ones. The best-known string matching algorithms are the Knuth-Morris-Pratt [11] and the Bayer-Moore ones [12].

The problem of subtree matching can be defined in several ways and it has a number of variations. The fundamental problem is the following: Let $G(V_G, E_G)$ and $H(V_H, E_H)$ be two trees. Let G be the subject tree and let H be the pattern tree. Now we want to find all the subtrees of G that matches tree H or decide that there is no such subtree in G. The original problem may vary upon two orthogonal things. They are: firstly the type of trees and secondly the exact definition of matching. Concerning the tree types trees can be node labeled or not node labeled, rooted and unrooted, ordered and unordered. The definitions are as follows:

Definition 1 (Labeled tree): Let Σ_V be a finite alphabet of vertex labels. Let V be a finite nonempty set of vertices, l a total function $l: V \to \Sigma_V$, E a set of unordered pairs of distinct vertices called edges. $G=(V, l, E)$ is a node labeled graph. A labeled tree is a connected acyclic labeled graph.

Definition 2 (Rooted tree): A tree is called a rooted tree if one vertex has been designated the root, in which case the edges have a natural orientation, towards or away from the root.

Definition 3 (Ordered tree): An ordered tree is a tree in which the children of each node of the tree are well-ordered.

The matching subtree condition can also be defined in several ways, resulting increasingly complex problems.

The simplest approach is the *subtree isomorphism problem* [1][2]: Given a subject tree G and a pattern tree H. Let us find all the subtrees of G that are isomorphic to H or decide that there is no such subtree in G. Graph G and graph G' (trees in this case) are said to be isomorphic if there is an f bijection between their vertices such that any two vertices u and v from G are adjacent if and only if $f(u)$ and $f(v)$ are adjacent in H. The basic algorithm for this problem was given by Matula [1] and Chung [6]. The time complexity of their algorithm is $O(k^{1.5}n)$, where k and n are the number of vertices in the pattern and in the subject tree, respectively. Tsamir and Tsur [2] improves this algorithm resulting an $O((k^{1.5}/\log k)\, n)$ time complexity.

The *subtree homeomorphism problem* [2][7] is a variation of the former problem, but instead of isomorphism homeomorphism must be checked. A homeomorphism or topological isomorphism is a special isomorphism between topological spaces which respects topological properties. This definition allows deleting degree 2 nodes from the subject tree and adding edges between their two neighbors to form a match with the pattern tree. Chung's previous algorithm [6] solves this problem as well.

A more advanced problem is the *tree inclusion problem*: Given a subject tree G and a pattern tree H locate the smallest subtree of G that includes H, where a tree T includes an other tree T', if T' can be obtained from T by deleting nodes of T. Fig. 1 shows an example for it. Unfortunately this problem for unordered trees is proven to be NP-hard by Kilpelainen and Mannila [15], however they showed a polynomial time algorithm that solves tree inclusion problem on ordered trees. Their algorithm uses $O(n_H n_G)$ time and space, where n_H and n_G are the number of nodes in H and G, respectively. The original algorithm was improved by Chen [16] who presented an algorithm using $O(l_H n_G)$ time and $O(l_H \min\{d_G, l_T\})$ space. Here l_S and d_S denoted the number of leaves and the maximum depth of tree S, respectively. Bille and Gørtz [17] were the first to show an algorithm with linear space bounds. With their algorithm tree inclusion problem can be solved in $O(\min\{n_H n_G/\log n_G,\ l_H n_G,\ n_H l_G \log\log n_G\})$ time using $O(n_H + n_G)$ space.

1-4244-9708-8/06/$20.00 ©2006 IEEE.

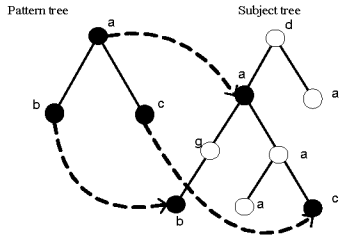

Figure 1
An example for the inclusion problem

The last variation of the subtree matching problem is the *pattern matching problem*. The problem is given by a subject tree S and a pattern tree P (or a set of pattern trees $P_1 \ldots P_k$). Both trees are rooted, ordered and labeled. Pattern trees can contain variable nodes labeled by the special letter *v*. They are the wildcards. A pattern tree P matches a subject tree S at node n if there are trees $t_1 \ldots t_k$ such that by substituting the *i*-th occurrence of *v* in P with t_i, the subtree of S rooted at node n can be obtained. For the solution of the problem all occurrences of all patterns should be located. Patterns can be linear and nonlinear. Nonlinear patterns do not contain any restrictions for the variables while nonlinear patterns may force some variables to be equal (the subtrees mapped to certain variables must be equal). Hoffmann and O'Donnell [20] make a thorough investigation into the problem. They provide a bottom-up algorithm which is the generalization of the Knuth-Morris-Pratt string-matching algorithm, and a top-down method that reduces a tree matching to a string matching problem. Both of the algorithms are made up by a preprocessing and a matching phase and they need a significant amount of space. The complexity of the naïve solution of the problem is O(nm) where n and m are the numbers of nodes in the subject and in the pattern tree, respectively. Till now the best known algorithm was given by Cole, Hariharan and Indyk [23]. The time complexity of their algorithm is $O(n\log^3 n)$.

II SUBTREE ISOMORPHISM

Problem definition: Given a subject tree G and a pattern tree H. Let us find all the subtrees of G that are isomorphic to H or decide that there is no such subtree in G.

The problem for general graphs if NP-complete, but for trees it can be solved in polynomial time [4].

A Naïve Algorithm for Rooted Trees

The basis of naïve solutions to subtree isomorphism problems is tree isomorphism. Once one can determine whether two rooted trees are isomorphic or not all the subtrees of the subject tree can be checked against the pattern tree resulting a naïve solution. We say that two rooted trees are isomorphic if there is an isomorphism between them such that their root nodes are mapped to each other.

An algorithm for deciding whether two rooted trees, T_1 and T_2 are isomorphic is given in [3]. It is based on the preposition that two trees are isomorphic if and only if one of them can be transformed into the other by permuting the child nodes at any node. This is a bottom-up algorithm that assigns integer values to every node of the two trees. At the end of the algorithm the trees proves to be isomorphic if and only if the root nodes are assigned the same number. The steps of the algorithm are as follows:

1 Assign the integer 0 to all the leaf nodes of T_1 and T_2.
2 At booth trees on level *i*-1 let us create a tuple for each nonleaf node. The items of a tuple are the numbers assigned to the child nodes (on level *i*) in not descending order. Let S_1 be the sequence of the (nonleaf) tuple on level *i*-1 in T_1 and let S_2 be the sequence of the (nonleaf) tuples on level i-1 in T_2.
3 Let us sort sequence S_1 and S_2, and let the outcome be S_1' and S_2', respectively.
4 If S_1' and S_2' are different T_1 and T_2 are not isomorphic. Otherwise let us label the nodes on level *i*-1 in both trees. Let us assign the integer 1 to all the nodes with the first distinguishable tuple, let us assign integer 2 to all the nodes with the second distinguishable tuple, and so on. Now all the nodes in both trees at level *i*-1 are labeled so we can move to level *i*-2 unless we reached the root node.
5 If the same integer value is assigned to the roots of T_1 and T_2 the two trees are isomorphic otherwise they are not.

Fig. 2 shows an example for labeling the trees. This algorithm can easily be extended to labeled trees as well. The only thing that should be changed is the method how a tuple is created: The label of a node should be inserted at the first position of the tuple, thus not only the child nodes but also the labels can make two nodes different. The complexity of the above algorithm is O(n) where n is the number of nodes in both T_1 and T_2.

Now it is easy to conceive a naïve tree pattern matching algorithm based on subtree isomorphism for rooted (labeled or nonlabeled) trees. Just traverse the subject tree in preorder and perform a comparison procedure for every node to test for possible occurrences of the pattern tree. In the worst case this naïve algorithm requires O(nm) time for matching.

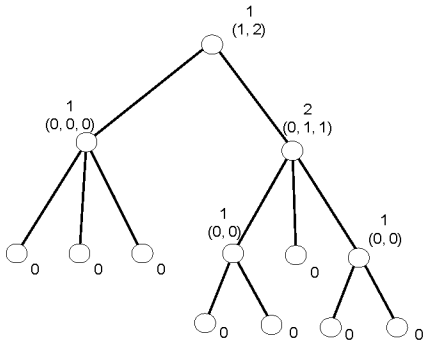

Figure 2
Labeling the nodes in the tree isomorphism problem

The best algorithm for rooted and ordered trees is O(nm/logm).

B Subtree Isomorphism for Unrooted Trees

The subtree isomorphism problem in general is not restricted to rooted trees. The basic algorithm for the general problem was given by Matula [1] and Chung [6] and was improved by Tsamir and Tsur [2]. The time complexity of Chung's algorithm is $O(k^{1.5}n)$. The algorithm

reduces the subtree isomorphism problem to the problem of maximum matching in bipartite graphs. The subtree isomorphism problem is recursively translated into a collection of smaller subtree isomorphism problems, which are solved using maximum matching algorithms. Tsamir and Tsur [2] improve the time complexity by performing the maximum matching in a simpler way.

For first we introduce some definitions. A rooted tree is a triplet $G = (V, E, r)$, where (V, E) is an unrooted tree, and r is some vertex in V which is called the root. G^r denotes that r is the root of the rooted tree G. We denote G_v^r the rooted subtree of G^r whose vertices are all descendents of v and its root is v. We write $H^S \subseteq_R G^R$ if there is a rooted subtree J^r of G^r which is isomorphic to H^S. We define the open neighbors of a vertex v in a graph G by $N(v) = \{u : uv \in E\}$ and the closed neighbors by $N[v] = \{u : uv \in E\} \cup \{v\}$. And $d_G(v)$ denotes the degree of vertex v in graph G (G can be omitted if it is clear from the context).

Now we will briefly describe the core of Chung's algorithm. Let $G = (V, E)$ be the subject tree and $H = (V_H, E_H)$ be the pattern tree. The algorithm selects a root node for G and lets H be unrooted. It will not restrict the solution of the problem. Let vertex r of G be the root of G. Now let us try to find a root node in H with which H will be an isomorphic tree with a subtree of G. Formally, we wish to know for each $v \in V$ and $u \in V_H$ whether $H^u \subseteq_R G_v^r$. It can be decided recursively, as $H^u \subseteq_R G_v^r$ if and only if for every child u' of u there is a distinct child v' of v such that $H_{u'}^u \subseteq_R G_{v'}^r$. It can be generalized in the following lemma:

Lemma: For every vertex v in G^r, vertex u in H, and a vertex w in N[u], we have that $H_u^w \subseteq_R G_v^r$ if and only if for every child u' of u in H_u^w there is a distinct child v' of v such that $H_{u'}^u \subseteq_R G_{v'}^r$.

Now form sets $S(v, u)$ for every $v \in V$ and $u \in V_H$ in the following way:

$$S(v,u) = \{w \in N[u] : H_u^w \subseteq_R G_v^r\} \quad (1)$$

Thus $S(v, u)$ contains all the adjacent nodes of u (also allowing u) in H that could be successfully used as root nodes in H for an isomorphic match between the subtree defined by node u in H and the subtree defined by node v in G. The algorithm calculates the sets $S(v, u)$ for all v and u. It is done in a recursive, bottom-up way starting with the leaf nodes:

- Leaf nodes: If u is a leaf of H, then $S(v, u)$ just contains one vertex which is the single neighbor of u in H. Otherwise if u is an internal vertex, and $S(v, u)$ is empty.
- Recursion: To calculate $S(v, u)$ for the internal vertices of G let us reduce the problem to a constraint for the child nodes. For an internal vertex v we first need to compute $S(v', u)$ for all children v' of v and for all $u \in V_H$. Now, to determine for all $w \in N[u]$ whether $w \in S(v,u)$, we construct a bipartite graph $B_w(v, u)$ that is made up by the child nodes of v and u. Formally, the two parts of $B_w(v, u)$ are $X_w^{v,u}$ and, $Y^{v,u}$ where $X_w^{v,u}$ is the set of children of u in H_u^w, $Y^{v,u}$ is the set of children in of v, and $u'v'$ is an edge of $B_w(v, u)$ if and only if $H_{u'}^u \subseteq_R G_{v'}^r$. The edges can be drawn on the basis of the previously computed sets for the child nodes. There is an edge between u' and v' if and only if $u \in S(v', u')$. Now, by the previous lemma w is in $S(v, u)$ if and only if $B_w(v, u)$ has a matching of size $|X_w^{v,u}|$. Therefore, in order to compute $S(v, u)$ we need to find maximum matching in d(u)+1 bipartite graphs.

The complexity of this algorithm is $O(k^{1.5}n)$ and uses O(kn) space. It was improved by Tsamir and Tsur by realizing that when computing $S(v, u)$ for an internal node instead of finding maximum matching in d(u)+1 bipartite graphs, it is enough to compute just the maximum matching of one graph due to the similarities of them. Since each graph $B_w(v, u)$ (for $w \neq u$) can be obtained by deleting vertex w from the graph $B_u(v, u)$. They also showed a faster maximum matching algorithm for this case. Thus, resulting an $O((k^{1.5}/\log k)\, n)$ algorithm.

III TREE INCLUSION

Unfortunately the tree inclusion problem for unordered trees is proven to be NP-hard by Kilpelainen and Mannila [15], however they also showed a polynomial time algorithm that solves tree inclusion problem on ordered trees. Tree inclusion problems have got great importance nowadays. They can be applied for retrieving data from XML documents [14], since an XML document can be viewed as a rooted, ordered and labeled tree. In the rest of this section we will only deal with rooted, ordered and labeled trees. The tree inclusion problem can be defined in two different ways, but resulting the same constraint:

Problem definition 1: Given a pattern tree P and a target tree T. Both are rooted, ordered and labeled. Tree P is included in tree T, denoted $P \subseteq T$ if P can be obtained from T by deleting nodes of T. Deleting a node v in T means making the children of v the children of the parent of v and then removing v. The children are inserted in the place of v in the left-to-right order among the siblings of v.

Before the second problem definition we introduce the concept of mapping:

Definition: For every node of a rooted ordered tree T let us assign numbers that represent the position in the preorder traversal. Let t[i] denote the node in T to which the number i is assigned. A mapping from a tree T_1 to a tree T_2 is a set M of ordered pairs of numbers (i, j), $1 \le i \le n_1$, $1 \le j \le n_2$, satisfying the following conditions, for all (i_1, j_1), (i_2, j_2) in M:

- $i_1 = i_2$ if and only if $j_1 = j_2$
- $label(t_1[i_1]) = label(t1[j_1])$ (label preservation condition)
- $t_1[i_1] < t_1[i_2]$ if and only if $t_2[j_1] < t_2[j_2]$ (order condition)
- $t_1[i_1]$ is an ancestor of $t_1[i_2]$ if and only if $t_2[j_1]$ is an ancestor of $t_2[j_2]$ (ancestor condition)

This mapping from T_1 to T_2 is also called embedding of T_1 in T_2 in some papers.

Problem definition 2: Given a pattern tree H and a target tree G. Both are rooted, ordered and labeled. Let us decide whether there is a mapping from H to G.

The two problem definitions above are equal. The first polynomial time algorithm was given by Kilpelainen and Mannila [15]. Their algorithm uses $O(n_H n_G)$ time and space, where n_H and n_G are the number of nodes in H and G, respectively. The original algorithm was improved by Chen [16] who presented an algorithm using $O(l_H n_G)$ time and $O(l_H \min\{d_G, l_G\})$ space. Here l_S and d_S denoted the number of leaves and the maximum depth of tree S, respectively. The main idea behind their algorithms is common. They traverses the target tree in a bottom-up way recursively starting at the leaves and try to match the nodes of the pattern tree with the nodes of the target tree based on the matching pairs one level lover in both trees. Formally: Let the pattern tree be $H(V_H, E_H)$ and let the target tree be $G(V_T, E_T)$. Let $v \in V_H$ and $w \in V_G$ be nodes with child node $v_1 \ldots v_i$ and $w_1 \ldots w_j$, respectively. To decide weather subtree $G(w)$ includes subtree $H(v)$ we should find a sequence $1 \le X_1 < X_2 < \ldots < X_i \le j$ such that $G(w_{xk})$ includes $H(v_i)$ for every $1 \le k \le i$. If we have already determined weather $G(w_t)$ includes $H(v_s)$ or not for every $1 \le t \le j$, $1 \le s \le i$, it is easy to find such a sequence. This sequence is basically a mapping from all the child nodes of v in the pattern tree to a distinct child node of w in the subject tree. Hence, applying this approach in a bottom-up fashion we can determine if $G(w)$ includes $H(v)$, for every $w \in V_G$, $v \in V_H$.

Bille and Gortz [17] were the first to show an algorithm with linear space bounds. They take a significantly different approach. The main idea of their algorithm is to create a data structure on tree T where certain operations called set procedures can easily be executed. Such set procedures are for example finding the nearest ancestor of two nodes or finding a deep set in a tree (nodes which are not ancestors of each other). They introduce the definition of deep embeddings: Let P be the pattern tree and T be the subject tree. An embedding f is said to be deep if there is no embedding g such that $f(root(P))$ is the ancestor of $g(root(T))$, where $root(G)$ denotes the root node of tree G. The deep occurrences of P in T, denoted $emb(P, T)$ is the set of nodes:

$$emb(P, T) = \{f(root(P)) \mid f \text{ is a deep embedding of } P \text{ in } T\} \quad (2)$$

Their algorithm is based on the fact that T includes P if and only if there is an embedding of P in T. And it is enough to find the deep embeddings, as if a subtree of T rooted at s includes P then all the ancestors of s will also include it. They showed an algorithm that computes $emb(P, T)$ using their set procedures. They provide several solutions with different time and space complexities. These algorithms only differ in the implementation of the data structure on tree T and the implementation of the set procedures. Their best result is $O(\min\{n_P n_T/\log n_T, l_P n_T, n_P l_T \operatorname{loglog} n_T\})$ time and $O(n_P + n_G)$ space complexity.

IV Tree pattern matching

For the tree matching problem one or more pattern trees should be located in a subject tree, such that all the trees are rooted, ordered and labeled and the pattern tree can contain wildcard leaf nodes. Now let us define it more precisely. For defining a tree we will use functions. We are given a finite ranked alphabet Σ of function symbols, including constants as nullary functions. S denotes the set of Σ-terms, formally defined as follows:

- For all b in Σ of rank 0, b is a Σ-term.
- If a is a symbol of rank q, then $a(t_1,\ldots,t_q)$ is a Σ-term provided that each t_i is a Σ-term.

Σ-terms can be viewed as labeled ordered trees. Term a(a(b, b),a) is shown in the Fig. 3 for an example. We are also given a special nullary symbol v, not in Σ, to serve as placeholder for any Σ-tree. S_v denotes the set of Σ u $\{v\}$-terms. A tree pattern is any term in S_v. A pattern tree P matches a subject tree T at node n if there are trees $t_1 \ldots t_k$ such that by substituting the i-th occurrence of v in P with t_i, the subtree of T rooted an n can be obtained.

The complexity of the naïve algorithm for matching a pattern in a subject tree is O(nm).

Hoffmann and O'Donnell [20] gave two general techniques for matching pattern trees $p_1 \ldots p_k$ in subject tree T. They presented a bottom-up and a top-down algorithm. The first one is the generalization of the Knuth-Morris-Pratt string-matching algorithm and the second one is the reduction of tree matching to a string matching problem. Both of them contain a preprocessing phase to achieve an efficient matching. The key idea of the bottom-up matching algorithm is to find, at each point in the subject tree, all patterns and all parts of patterns which match at that point. During the preprocessing phase matching sets are computed. A matching set contains all the subtrees of the pattern trees that matches the subtree of the subject tree rooted at a certain point. All of these matching sets are provided with a code. Each leaf is then assigned to a code corresponding to the matching set that contains the constant symbol at the leaf. Each function symbol is provided with a table which dimension is the arity of the function. This table tells the resulting matching set code on the basis of the matching set codes at the child nodes. If the matching set contains a whole pattern than a pattern occurrence is found. The complexity of the matching algorithm is O(n) as only the new code should be computed at each nodes of the subject tree. The main drawbacks of this method are the exponential size of the tables and thus the preprocessing time for computing them.

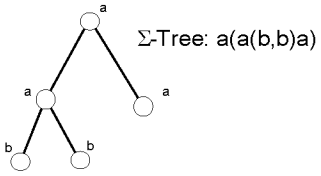

Figure 3
An example for a Σ-tree

The top-down method has slower matching time than the bottom-up, but faster preprocessing time. The key idea in reducing tree pattern matching to string-pattern matching is to represent each root-to-leaf path of the pattern tree with a string. The string will contain the function symbols and the numbers of edges that will indicate which branch from father to son has been followed. Thus the problem is reduced to string matching. We can use the multiple string matching algorithm of Aho and Corasik [24] to create an automaton that will recognize every path string of the pattern. Then we traverse the subject tree in preorder and

also compute the automaton states as we visit nodes and traverse edges. We should also count the occurrences of the path strings during the traverse of the subject tree, and when all of them were detected a match is found. The time complexity of the preprocessing phase (creating the automaton) is O(m), where m is the number of nodes in the pattern tree and the time complexity of the matching phase is somewhere between the naïve algorithm and the bottom-up method. The worst case matching time of the algorithm is $O(nl^*)$, where l^* is the maximum suffix number in the set of path strings of the pattern node.

Till now, the fastest algorithm for pattern matching was given by Cole, Harisharan and Indyk [23]. They reduce the tree matching problem to the subset matching problem. The time complexity of their algorithm is $O(n\log^3 n)$.

Tree pattern matching can also be solved for nonlinear patterns. A solution is shown by Ramesh and Ramakrishnan [22]. Their algorithm uses O(m) for preprocessing and $O(nk^*)$ for matching where k is the number of root-to-leaf path string whose leaves are labeled with variables ($k \leq 1$).

V SUMMARY

Table 1 summarizes the algorithms covered in this paper. It shows the algorithms categorized and chronologically and reveals the invertors and their time complexities.

VI CONCLUSION

This paper has given an overview about the different kind of problems that are related to subtree matching. It classified the matching problems into three categories: subtree isomorphism, subtree inclusion that is an extension of subtree homeomorphism and subtree matching. It introduced the basic algorithms for these problems and showed the currently best algorithms for them. Though there are some problems related to the problem of subtree matching that are not covered in this paper. The most important ones are the different kinds of distances between two trees that can be learned from [25], the maximum common embedding subtree problem discussed in [26], and the tree canonization [5][8].

Table 1
The most important algorithms and their time complexities related to subtree matching

Algorithm	**Time complexity**
Graph isomorphism	Unknown (GI-hard)
Subgraph isomorphism	NP-hard
Subtree isomorphism	
Naïve for rooted trees [3]	O(nm)
For rooted and ordered trees	O(nm/log m)
Matula (1978) [1]	$O(n^{5/2})$
Shamir, Tsur (1999) [2]	$O((k^{1.5}/\log k)\, n)$
Subtree homeomorphism	
Chung (1987) [6]	$O(N^{2.5})$
Tree inclusion	
For unordered trees [15]	NP-hard
Kilpelainen and Mannila (1995) [15]	$O(n_H n_G)$
Chen (1998) [16]	$O(l_H n_G)$
Bille, Gørtz (2005) [17] (also improves space usage)	$O(\min\{ n_P n_T/\log n_T, l_P n_T, n_P l_T \text{loglog } n_T \})$
Tree pattern matching	
Naïve algorithm	O(nm)
Hoffmann, O'Donell (1982) [20] Bottom-up	O(n) + exponential preprocessing
Hoffmann, O'Donell (1982) [20] Top-down	$O(nl^*)$ + O(m) preprocessing
Dubiner, Ganin, Magen (1994) [21]	$O(nm^{0.5}\log m)$
Ramesh, Ramakrishnan (1992) for nonlinear patterns	$O(nk^*)$ + O(m) preprocessing
Cole, Harisharan, Indyk (2000) [23]	$O(n\log^3 n)$

REFERENCES

[1] D. W. Matula, "Subtree isomorphism in $O(n^{5/2})$ – Matula," Annals of Discrete Mathematics, Vol. 2, 1978, pp. 91-106

[2] R. Shamir and D. Tsur, "Faster Subtree Isomorphism," J. Algorithms, Vol. 33, No. 2, 1999, pp. 267-280

[3] A. V. Aho, J. E. Hoprcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, NJ: 1982, p. 98

[4] A. Gupta and N. Nishimura, "The Complexity of Subgraph Isomorphism: Duality Results for Graphs of Bounded Path- and Tree-Width," Apr. 1995.

[5] Dinitz, Itai and Rodeh, "On an Algorithm of Zemlyachenko for Subtree Isomorphism," IPL: Information Processing Letters Vol. 70, 1999

[6] M. J. Chung, "O(n2.5) Time Algorithms for the Subgraph Homomorphism Problem on Trees," J. Algorithms, Vol. 13, 1987, pp. 106-112

[7] Pinter, Rokhlenko, Tsur and Ziv-Ukelson, "Approximate Labelled Subtree Homeomorphism," in Proceedings of 15th Symposium on Combinatorial Pattern Matching, 2004, pp. 59-73

[8] Samuel R. Buss, "Alogtime Algorithm for Tree Isomorphism, Comparison and Canonization," in Proceedings of Kurt Gödel Colloquium, 1997, pp. 18-33

[9] D. Eppstein, "Subgraph Isomorphism in Planar Graphs and Related Problems," in Proceedings of SODA, 1995, pp. 632-640

[10] M. R. Garay and David S. Johnson, Computers and intractability : a guide to the theory of NP-completeness, W.H. Freeman, 1979

[11] Knuth, Morris and Pratt, "Fast Pattern Matching in Strings," SICOMP: SIAM Journal on Computing, Vol. 6, 1977

[12] Robert S. Boyer and J. Strother Moore, "A Fast String Searching Algorithm," Commun. ACM, Vol 20. No. 10, 1977, pp. 762-772

[13] Gyula Katona, András Recski and Csaba Szabó, "Gráfelmélet, algoritmuselmélet és algebra," NJ: 1997

[14] Miklau and Suciu, "Containment and Equivalence for Fragment of XPath," JACM: Journal of the ACM, Vol. 51, 2004

[15] Kilpelainen and Mannila, "Ordered and Unordered Tree Inclusion," SICOMP: SIAM Journal on Computing, Vol. 24, 1995

[16] W. Chen, "More Efficient Algorithm for Ordered Tree Inclusion," Journal of Algorithms, Vol. 26, No. 2, 1998, pp. 370-385

[17] Bille and Gortz, "The Tree Inclusion Problem: In Optimal Space and Faster," in Proceedings of Annual International Colloquium on Automata, Languages and Programming, 2005

[18] C. M. Hoffmann and M. J. O'Donnell, "An Interpreter Generator Using Tree Pattern Matching," in Proceedings of Principles of Programming Languages *(POPL'79)*

[19] Min Zhao and Sachin S. Sapatnekar, "A New Structural Pattern matching Algorithm for Technology Mapping," in Proceedings of the 2001 Design Automation Conference ({DAC}-01), pp. 371-376

[20] M. Hoffmann and M. O'Donnell, "Pattern Matching in Trees," Journal of the Association for Computing Machinery, Vol. 29, 1982, pp. 68-95

[21] Dubiner, Galil and Magen, "Faster Tree Pattern Matching," JACM: Journal of the ACM, Vol. 41, 1994

[22] R. Ramesh and I. V. Ramakrishnan, "Nonlinear Pattern Matching in Trees," Journal of the ACM, Vol. 39, No. 2, Apr. 1992, pp. 295-316

[23] R. Cole and R. Hariharen, "Tree Pattern Matching and Subset Matching in Deterministic O(nlog3n)," in Proceedings of the Tenth Annual {ACM}-{SIAM} Symposium on Discrete Algorithms, Jan. 1999, pp. 245-254

[24] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," Comm. A.C.M., Vol. 18, No. 2, Jun. 1975, pp. 333-340

[25] Gabriel Valiente, "An Efficient Bottom-Up Distance Between Trees," in Proceedings of SPIRE, 2001, pp. 212-219

[26] Anonio Lozano and Gabriel Valiente, "On Maximum Common Embedded Subtree Problem for Ordered Trees," String Algorithms, 2004, pp. 155-169

[27] Pinter, Rokhlenko, Tsur and Ziv-Ukelson, "Approximate Labeled Subtree Homeomorphism," in Proceesings of 15th Symposium on Combinatorial Pattern, 2004